TLNK Transputer Linker User Guide

TLNK Version 91.1
6/15/91
Copyright 1987-1991 by Logical Systems

Contents

1 Introduction
- Overview
- Virtual Memory Option: VTLNK
- System Requirements

2 Usage
- Getting Started
- Using TLNK Interactively
- Using TLNK With a Command File
- Operational Statistics
- Listing File Description

3 Appendix A: Error Messages
- Types of Error Messages
- Error Message Descriptions

4 Appendix B: TLNK Internals
- Source Code Organization and Compiling

Introduction

Overview

TLNK is a linker designed for use with INMOS Transputers. It supports multiple libraries and allows complete control of load addresses for each program or library code fragment. TLNK is designed for use with the TASM assembler and TLIB librarian. It is also the linker used with the TCX Transputer "C" compiler system. TLNK may be run either as an "interactive" program or may be run in "batch" mode with a user specified command file.

The architecture of the Transputer requires that some of the code generation be delayed until the linker/locater stage to insure minimum length prefix strings are generated for all instructions. TLNK supports this by processing information which TASM puts in the relocatable files to determine which instructions have been completed at assembly time and which TLNK will have to finish during the linkage operation.

TLNK uses a multiple pass algorithm to minimize instruction prefix lengths. The algorithm used doesn't guarantee minimum length prefixes in all cases (generating a minimum length program is a theoretically "hard" problem), but does a pretty fair job in a moderate amount of time.

TLNK is a single pass, file based, linker. It uses temporary files to hold the intermediate results of the linkage process. The only limitations on the size of what can be handled are related to available disk space and data memory size (if extreme numbers of symbols are used). As released, TLNK will support 2048 external symbols and 5450 total symbols (the sum of the external and exported local symbol references, and definitions, in all the input and library files used). If TLNK is recompiled and used on a machine with a larger direct address space, these limits can be greatly increased. The limits can also be increased under MS-DOS by using a mixed memory model (automatically handled during the compilation of TLNK when changes in the limits require it).

Virtual Memory Option: VTLNK

As an option for machines which support virtual memory (or have enough directly addressable physical memory), TLNK may be compiled to use memory in lieu of temporary files. This option speeds up the linking process and is particularly beneficial for large programs consisting of many modules. See the **Source Code Organization and Compiling** section of **Appendix C** for more information about configuring TLNK to use memory instead of temporary files. The VTLNK program provided in the distribution is a version of TLNK which has been configured in this fashion. Usage of VTLNK will be limited to small or medium sized applications on machines (or operating systems), which only support a 32K or 64K direct addressing range.

System Requirements

The file based version of TLNK requires approximately 256K of program memory space to run. It should run in any environment which supports other major system development tools (compilers, etc.). As mentioned, it uses temporary files for holding intermediate results, requiring file space equal to about 150% of the sum of the input relocatable files and whatever files are used out of libraries. TLNK also requires an additional 100% worth of file space to hold the linked output file. TLNK makes heavy use of these temporary files and provisions have been made to allow the user to select where the temporary files will be located (ideally some sort of ram-disk). TLNK opens a separate temporary file for each "module" present in one or more input or library files. It may also have as many as 5 additional files open to support the normal linkage process (input, output, map, etc.). If you use a large number of modules you should ensure that your operating system will allow the requisite number of files to be opened at the same time.

Usage

Getting Started

The general form of the TLNK command line is:

```
tlnk [command_file]
```

If you provide a "command_file", TLNK will read it for commands. If you don't provide this file, TLNK will prompt you directly for what to do. The format for commands on a "command_file" is similar to what TLNK prompts you with, and expects in return, when running interactively.

Using TLNK Interactively

The following describes what TLNK asks for when you are running interactively:

• `"Flags (A|a|C|c|S|s) [none]:"`. This is the first question TLNK asks. It's designed to serve the same function as command line option flags do with other programs (TASM, TLIB, etc.) This question is answered by typing zero or more of the supported flags, followed by a carriage return. The flags are case insensitive. Simply typing a carriage return tells TLNK that you do not wish to use any of the options. Note that the value in brackets mentioned in the questions TLNK asks indicates what default action will be taken if you respond only with a carriage return. Each of the three possible option flags has the following effect:

> A - Sort the symbol table produced as part of the TLNK listing file in increasing address order. If you don't specify this flag the symbol table will be sorted in increasing alpha order. Note that you must specify a listing file (see below), for this flag to have any effect.
>
> C - Remove debugging information from the input files and compress the intermediate and output files during linking. This can provide a substantial reduction in execution time for TLNK (assuming the debugging information won't be needed).
>
> S - Link the code assuming the "static-link" global data model is desired. See the TCX documentation for more information about what all this entails.

• `"Temp file path [.]:"`. This is the second question TLNK will ask. As mentioned, TLNK does a lot of temporary file I/O, and this question allows you to specify where the temporary files should be placed. Your answer should specify a directory on the fastest file I/O device which is both available, and large enough, to hold all the temporary files needed. As an alternative, if you answer with a carriage return, TLNK will put the temporary files on the directory associated with the "TMP" environment variable if it exists, or else the current directory (represented by the "." inside the square brackets in the question TLNK asked). TLNK will automatically delete whatever temporary files it creates when it finishes. If you are running a virtual memory version of TLNK you may just ignore this question (answer it with a carriage return).

• "Listing file [NUL]:". By default TLNK does not create a listing/map file of what it does. If you wish one, provide a file pathname in response to this question. If not, answer with a carriage return. You should make sure that any filename you give has any desired extension, since TLNK does not provide one by default.

• "Input files(s) [.trl]:". You are required to provide at least one input file to be linked in response to this question. Note that the default within the brackets means that a filename extension of ".trl" will be assumed for all filenames which lack explicit extensions. You may link more than one file together by separating them with commas. Some example responses:

```
"test" or "test.trl"
```

These would tell TLNK to link a single file named "test.trl".

```
"test,test1,test2.zip"
```

This would tell TLNK to link three files: "test.trl", "test1.trl" and "test2.zip". If you have more files than will fit on the line in response to the "Input file(s) [.trl]:" question, you may enter as many as you like on the line followed by a trailing comma, and TLNK will prompt you for additional input files. You may also use spaces as delimiters (instead of commas), but trailing spaces will NOT cause TLNK to prompt you for another line of input files.

TLNK also provides the ability to individually specify the starting load address for any "module" within any input file (see the TASM manual for a description of the ".mod" pseudo-op, and modules in general). The load address must be word aligned, but may otherwise be any desired value. In the above examples no load address was specified for the input files. To specify a load address for a module the following command is given:

```
module_number ":" load_address
```

This load address holds for all instances of the specified module which follow it in the list of input files you give TLNK. It will also affect the load address of higher numbered modules in ANY input file (either before or after the load address specification in the input file list you give TLNK). This is due to the load order of modules and files:

Modules are loaded starting with low numbered modules and working up in memory. Within a module, code and data fragments are loaded in the order the files they are contained in appear in the input file list you give TLNK. The load address for the lowest numbered module is fixed by the response you give to the "Load address [0x????????]:" question TLNK asks (described later in this section), unless you provide an explicit load address BEFORE the first occurence of the lowest numbered module. If you specify an explicit load address, all remaining code/data in that module will be affected, AS WELL AS, code/data in any higher numbered modules which aren't provided with explicit load addresses. Thus, if module #7 is to be loaded at address 0x00001000 ("C" style hexadecimal constant), the input file list might look like:

```
7:0x1000,test
```

This would cause any code or data in module #7 (of "test.trl"), to be loaded starting at address 0x1000. Because of load order, any higher numbered modules in "test.trl" would be loaded following module #7 in memory, NOT following whatever the effective load address was after TLNK processed any lower numbered modules in "test.trl". Note that the load address specification is separated from filenames and other load addresses by commas.

Suppose the load address was needed with file "test.trl", but "test1.trl" and "test2.zip" were to be loaded in the normal manner (except for any modules numbered higher than 7 which they contain):

```
test1,test2.zip,7:0x1000,test
```

Note that load addresses may be specified for modules which don't appear in any of the files in the input file list. This does no harm other than opening an extra temporary file. Module numbers range from 0 to 255 (decimal), but may be specified in decimal, hex or octal as desired ("C" style numeric constants). Likewise, load addresses may be specified in decimal, hex or octal form.

To take best advantage of the explicit load address capability, it is generally useful to make those modules which need to be located somewhere "special" be at the top of the module numbering scheme. Since an explicit load address for a module will override the current load address, all the "special" modules (with explicit load addresses), will be loaded where desired.

Note that there is NO checking of load address conflicts by TLNK. If you specify load addresses which conflict with each other (or specify a load address which conflicts with code or data whose load address is automatically determined by TLNK), you are going to have a problem. This can be detected by having TLNK generate a listing file which will contain a "map" of where all the modules have been allocated.

A final example:

```
test,1:1024,2:2048,test1,1:0800,test2.zip
```

If "test.trl" contains anything in modules #0 or #1, it is loaded at the default load address (#0 first). If "test1.trl" contains anything in module #1 it is loaded at address 1024 (decimal). If "test2.zip" contains anything in module #1 it is loaded starting at address 800 (octal). In turn, if "test.trl" contains anything in module #2, it is loaded at address 800 (octal), following anything "test2.zip" had in module #1. Finally, everything in modules #2 through #255 in files "test1.trl" and "test2.zip", and in modules #3 through #255 in "test.trl", is loaded at address 2048 (decimal), in normal load order. This appears complicated at first, but is pretty straightforward if you keep the load order in mind.

If you make a mistake on any of responses you give to the "Input file(s) [.trl]:" question TLNK will print an error message, throw away whatever input file information it has accumulated and ask you to reenter the input files (and explicit load addresses), from scratch. One of the mistakes detected in this fashion is the multiple definition of an external value. Although many linkers treat this as a warning and pick one of the definitions to use, the Transputer architecture makes this ill-advised. This is due to the possibility that a local reference to one of the external definitions may have already been satisfied during the assembly process, while the linker might end up choosing another definition later on. Having external references be variable length has interesting ramifications! Undefined external references are treated in a more normal fashion with a warning message being generated and zero substituted for the unknown value.

• "`Entry point [_main]:`". This is the third question TLNK asks. Your answer indicates where the program you are linking should begin executing. By default, TLNK assumes you will have a symbol named "_main" which it will use (the TCX "C" compiler generates this entry function for you). You may specify another symbol as the entry point, or specify an explicit memory address as the entry location if you desire. If you wish to specify a memory address you answer the question with a numeric address (in decimal, or "C" style hexadecimal or octal). If the first character of your answer is not a digit, TLNK assumes you are providing an entry point symbol name.

• "`Library file(s) [.tld]:`". This question allows you to indicate which libraries you wish to search to resolve undefined externals. You answer this question using the same syntax as used with the input files list in question #3. Explicit load addresses specified with libraries have the same effect on equal or higher numbered modules as with input files, and affect both modules in files from libraries and higher numbered modules in input files. Although not mentioned at the time, explicit load addresses in the input file list WILL affect equal or greater numbered modules in files read from libraries.

The libraries are searched in the order given in your response using a single pass algorithm. The order of files within your libraries may be important if library files also have external references which need to be satisfied by definitions in subsequent files in the library. You may also list a single library file more than once in your library file list if you wish to have a library searched more than once (to handle dependency "cycles" perhaps).

The directory that libraries must reside in is controlled by the use of an environment variable named "TLIB". This variable may be assigned a ';' separated list of directories to search when hunting for a particular library. By default, the current directory is first searched to find the desired library, then any directories specified by the environment variable are checked.

Additional information about libraries can be found in the "TLIB Librarian User Guide".

• "`Output file [????????.tld]:`". This question sets the output file to be used for the linked program. The "????????.tld" default is constructed using the name of the first input file with a filename extension of ".tld". The default is selected in the usual fashion (with a carriage return), anything else entered in response to this question is assumed to explicitly name the desired output file.

• "`Load address [0x????????]:`". This question allows you to set the initial load address for the program in the absence of explicit load address instructions in the input files, or libraries, specification. The default address for this question is dependent on which Transputer the program is being assembled for, but is always set to be the first address beyond on-chip memory. For more information about how load addresses work, see the description of the "`Input files(s) [.trl]:`" question (#4). If you provide a non-default response you may use either a decimal, hexadecimal, or octal number for the address ("C" style syntax). Note that since TLNK is designed to handle both 16 and 32 bit Transputers the legal addressing range for a 16 bit Transputer is 0xFFFF8000 to 0x00007FFF, NOT 0x8000 to 0x7FFF!

• "`Stack address[0x????????]:`". This is the last question TLNK asks you. Your response controls what address the workspace (stack) pointer is initialized to. The default answer to this question is dependent on which Transputer the program is being linked for, but is always set to be the first address beyond on-chip memory (the same as the default load address). It is assumed that the stack will grow downward, thus the first data written into the stack will be at the top end of on-chip memory. If the Transputer internal memory is too small to hold the required stack, the workspace/stack pointer may be set to be somewhere else (such as the top of external memory). If you provide a non-default response you may use either a decimal, hexadecimal, or octal number for the address ("C" style syntax).

Using TLNK With a Command File

The same basic questions/responses are used in a command file as when TLNK is used interactively. The command file is a simple ASCII text file with one TLNK question and response per line. Each line begins with a command which corresponds to one of the questions TLNK would ask you if run interactively, the remainder of the line contains the answer you would have given. For the sake of brevity the command/question portion of the line may be abbreviated. Note that you may include a comment in a command file by simply starting a line with a non alphanumeric character such as ';'. The following list shows the abbreviations and the questions they correspond to in interactive mode:

```
FLAG      "Flags(A|a|C|c|S|s) [none]:"
TEMP      "Temp file path [.]:"
LIST      "Listing file [NUL]:"
INPUT     "Input files(s) [.trl]:"
ENTRY     "Entry point [_main]:"
LIB       "Library file(s) [.tld]:"
OUTPUT    "Output file [????????.tld]:"
LOAD      "Load address [0x????????]:"
STACK     "Stack address[0x????????]:"
```

The same defaults apply here as in interactive mode, however, you may completely omit commands for which you wish to use the default. You must also ensure that you order the commands in the command file in the same order as the questions would have been asked interactively (also the order they are listed above). You may use a more complete command name than the abbreviations listed, but the name must start with the same characters as the abbreviation, and must contain no whitespace (spaces, tabs, etc).

The "answer" portion of the command line will be in the same format as if you were answering the question interactively. You must provide some whitespace between the command/question and the corresponding "answer". The following is an example set of commands for a TLNK command file:

```
LIST      test.map
INPUT     test
```

This would link a program named "test.trl", generate a link map on file "test.map", and output the resulting linked program to "test.tld" (by default). If you wished to search a library named "testlib.tll", and set the initial load address to zero, you might use:

```
LIST      test.map
INPUT     test
LIB       testlib
LOAD      0
```

The same explicit load addresses may be used in the INPUT and LIB commands as are allowed in the corresponding answers when running TLNK interactively.

Operational Statistics

If TLNK is being controlled from a command file, it doesn't write anything to standard output unless errors or undefined symbols are detected. If TLNK is being run interactively, it will write errors, undefined symbols, AND an operational summary, to standard output. The operational summary consists of information on how many symbols TLNK dealt with, and how many undefined symbols and errors it encountered.

The symbol information shows how many "external" symbols were processed and what percentage of the "external" symbol table was used doing so. In addition, the number of "total" symbols used is displayed, along with the corresponding percentage of the "total" symbol table capacity which was used. The contents of the "total" symbol table include ALL the local symbols which were exported to TLNK in all the input and library files, as well as the external references and definitions each input or library file contained. Note that references to the same external symbol in different input or library files occupy different slots in the "total" symbol table (a consequence of the multi-level symbol table structure used).

Listing File Description

If you specify a listing file, TLNK will write out a commentary on what it does, consisting of a listing of which files it reads, what is defined and referenced in those files, an address map of where the various "modules" are loaded, and a listing of the external symbols used and the associated values. The listing will also contain the same summary information which is written to standard output when TLNK is being run interactively. Examining it in more detail, the listing file consists of three separate sections:

- The first section shows which input files TLIB read, which files within the various libraries were needed, and which external symbols were defined or referenced in each file. The referenced and defined symbol names are listed in table form below the name of each file processed. If the symbol begins with a "*" the symbol was referenced, otherwise the symbol was defined in that file.

- The second section consists of a map of where the various "modules" of the program will be loaded in memory (see the "TASM User Guide" for a description of "modules" and the ".mod" pseudo-op). This section is organized as a list of all the modules within the program (in increasing order). Within a given module there may be multiple, discontinuous, regions, as a result of explicit load addresses specified in TLNK input or library file specifications. Each region description for a module is put on a separate line and indented for clarity. Each region description contains the starting address, ending address, and the size of the region. In a similar fashion, the sizes of all the regions within a module are summed to give a total module size. This section of the listing will NOT be present if TLNK detects any errors (the determination of actual addresses is generally impossible when errors are present).

- The final section of the listing consists of a symbol table for the entire program showing all the external symbols and their values. If a symbol was undefined its value is replaced with "********" in the listing (zero is assumed for the value). If any errors are detected by TLNK, the values of the various symbols will be only approximate, since accurate address binding may be impossible. After the symbol table the starting load, stack, and entry point addresses are listed, followed by the same information about errors and symbol table usage which TLNK generates when being run interactively.

Appendix A: Error Messages

Types of Error Messages

All error messages are written to standard output. If a listing file is being generated the error message is also written on it. There are three classes of error messages which TLNK can generate:

- **Warnings**. These are used to report problems which aren't severe enough to cause TLNK to abort (exit with a non-zero return value). These messages usually indicate trouble which isn't immediate, but may be soon! The format for warnings is:

```
WARNING: message_text
```

- **Non-fatal errors**. These are used for reporting actual error conditions which will affect the return value given when TLNK exits. If one or more non-fatal errors are encountered TLNK will return a non-zero return code, otherwise it will give a return code of zero. Another result of encountering non-fatal errors is that the generation of a linked output file is inhibited. The format for non-fatal errors is:

```
<filename> @ line_number: message_text
```

Where the "<filename>" field indicates the source code file which contained the problem, the "line_number" field gives the line where the problem was detected, and the "message_text field indicates the actual problem encountered. If the problem was not in one of the input or library files, but in a command file being used by TLNK, the filename and line number will refer to the appropriate command file instead.

- **Fatal errors**. If the problem detected by TLNK is so severe that it can't continue operating, it will give a "fatal" error message:

```
FATAL: message_text
```

After printing one of these messages, TLNK will immediately exit with its error return code set (non-zero).

Error Message Descriptions

The following descriptions list the various error messages which TLNK can generate (in alphabetic order). Note that error messages which have a trailing "filename" will display the name of the actual file involved:

```
*** At least one input file is required
```

TLNK requires at least one input file to link. You may correct this error by re-entering the input file specification list from the beginning and including at least one input file.

```
*** Bad record format on input file: filename
```

This error message is generated whenever a record in a input file doesn't conform to a legal record type. This indicates that the named file has been corrupted, or that a change made to the record format generated by TASM or TLIB has not been made to TLNK (for those who can't help doing a little "improving" to programs now and then). You may recover from this error by re-entering the input file specification list, omitting the file with the bad record. Since you normally need to link all the code and data which comprise a program you will probably have to re-create the bad file.

```
*** Illegal input filename
```

A input file pathname you provided in a TLNK input file specification list was malformed (either zero length or too long). As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). You may correct this error by re-entering the input file list from the beginning.

```
*** Illegal library filename
```

A library file pathname you provided in a TLNK library file specification list was malformed (either zero length or too long). As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). To correct this error you will have to run TLNK over from the beginning.

```
*** Illegal module number
```

TLNK gives this error when you have specified a module number outside the range of 0 to 255. This particular version of the error indicates you are running TLNK interactively. If this error was encountered on the input file specification list you will be allowed to re-enter the specification list. If this was encountered in the library specification list you will have start over and re-run TLNK.

```
*** Illegal numeric constant
```

This error message means that an incorrectly formatted number was encountered on the last information you provided to TLNK (which is being run interactively). Note that numbers may be decimal, octal, or hexadecimal ("C" style numeric constants).

```
*** Input file has already been linked: filename
```

TLNK doesn't allow files which have already been linked to be re-linked. Linked files have a default filename extension of ".tld". To correct this, re-enter the input file specification list from the beginning, omitting the already linked file. This error is detected by examining tag data in the input file to determine file type (thus, this error could also indicate a corrupted input file).

```
*** Input filename too long
```

A file pathname you provided in a TLNK input file specification list is too long. As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). You may correct this error by re-entering the input file list from the beginning.

```
*** Invalid flag character: character
```

The specified character wasn't recognized by TLNK as one of the option flags it supports. Either re-enter the desired option flags or type a carriage return to take the default (no options selected).

```
*** Libraries aren't legal input files: filename
```

The specified library file was named in the preceeding input file list. TLNK only allows normal relocatable files to be listed there (files with default extensions of ".trl"). To correct this, re-enter the input file specification list from the beginning, omitting the library file. This error is detected by examing tag data in the input file to determine file type (thus, this error could also indicate a corrupted input file).

```
*** Library filename too long
```

A file pathname you provided in a TLNK library file specification list is too long. As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). To correct this error you will have to run TLNK over from the beginning.

```
*** Load address not word aligned
```

The load address portion of a "module" specification, on a input or library file list, must be word aligned. For the case of the 32 bit transputers this means that the bottom two bits of the address MUST be zero. This error message is printed on the line following where the error was detected. If the error occurred in a input file list you should re-enter the input file specifications from the beginning. If this error occurred in the library file list you will have to run TLNK over from the beginning.

```
*** Missing ':'
```

This error indicates that your previous response to TLNK contained a module specification which was missing a ":" after the module number. If your response was a input files specification you will be allowed to re-enter the input file specification (from the beginning). If your response was a library file specification you will have to start TLNK over from scratch. See the **Using TLNK Interactively** section for a description of what a "module" specification looks like and does.

```
*** Unable to mix code for different CPU types
```

TLNK is unable to mix code assembled for different Transputer types (T414B, T800, etc). TLIB uses information provided in the various input files to ensure that all the code involved is for the same type of processor. To recover from this error you should re-enter the input files list, including only files assembled for one particular CPU type.

```
*** Unable to open input file: filename
```

TLNK was unable to open the specified input filename. You may correct this error by re-entering the input file specification list from the beginning and correcting the error in the filename. You should ensure that the filename (and associated optional directory path), were correctly typed noting that the default filename extension is ".trl" unless you explicitly specify one.

```
*** Unable to open output file: filename
```

The link output file you specified couldn't be opened. This could be caused by a bad filename, write-protected or full storage device, or other file system related problems.

```
<command_filename> @ line_number: Bad command ordering
```

TLNK gives this error when the command file you are using has commands in the wrong order. The desired order is the same as TLNK uses when it is being run interactively. See the **Using TLNK with a Command File** section for more information about command ordering.

```
<command_filename> @ line_number: Duplicate entry point
```

TLNK gives you this error when you provide more than one "ENTRY" command in a command file. The line number and filename of the offending command file are indicated.

```
<input_filename> @ line_number: Duplicate external definition: symbol
```

The named symbol at the specified point in the specified input file was a duplicate external symbol definition. Unlike some linkers which treat this as a warning, TLNK calls this an error since it has no way of knowing which (if any), of the symbol definitions were not used by TASM to locally bind a symbol reference (and must be kept).

```
<command_filename> @ line_number: Illegal input filename
```

A input file pathname you provided in a TLNK input file specification list was malformed (either zero length or too long). As released, TLNK allow pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). The name and line number of the command file where the error was detected are provided.

```
<command_filename> @ line_number: Illegal library filename
```

A library file pathname you provided in a TLNK library file specification list was malformed (either zero length or too long). As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). The name and line number of the command file where the error was detected are provided.

```
<command_filename> @ line_number: Illegal module number
```

TLNK gives this error when you have specified a module number outside the range of 0 to 255. This particular version of the error indicates a command file was being used (and the associated line number where the problem was encountered).

```
<command_filename> @ line_number: Illegal numeric constant
```

This error message means that an incorrectly formatted number was encountered on the specified line of the specified command file. Note that numbers may be decimal, octal, or hexadecimal ("C" style numeric constants).

```
<command_filename> @ line_number: Input filename too long
```

A file pathname you provided in a TLNK input file specification list is too long. As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). The name and line number of the command file where the error was detected are provided.

```
<command_filename> @ line_number: Library filename too long
```

A file pathname you provided in a TLNK library file specification list is too long. As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h"). The name and line number of the command file where the error was detected are provided.

```
<command_filename> @ line_number: 'LIST' filename already specified
```

You may only specify the "LIST" filename once in a command file. This error gives the name of the command file and the line number of a duplicate "LIST" command.

```
<command_filename> @ line_number: Load address not word aligned
```

The load address portion of a "module" specification, on a input or library file list, must be word aligned. For the case of the 32 bit transputers this means that the bottom two bits of the address MUST be zero. This error message gives the name and line number of the command file where this error was detected.

```
<command_filename> @ line_number: Missing ':'
```

This error indicates that on the specified line in the specified command file a "module" specification was missing a ':' after the module number. See the **Using TLNK Interactively** section for a description of what a "module" specification looks like and does.

```
<command_filename> @ line_number: No input file(s) specified
```

You failed to specify one or more "INPUT" filenames in the indicated command file. The line number reported may be ignored (it will be set to the end of the command file).

```
<command_filename> @ line_number: Unable to open output
file: filename
```

The link output file specified in the command file "OUTPUT" statement couldn't be opened. This could be caused by a bad filename, write-protected or full storage device, or other file system related problems.

```
FATAL: Address out of 16 bit addressing range
```

This error message is generated when TLNK is binding addresses to symbols for a 16 bit Transputer and it detects an address which is outside of the legal 0xFFFF8000 to 0x00007FFF addressing range. This error is usually caused by specifying an incorrect ENTRY, LOAD or STACK value to TLNK. It may also be caused by a program which is too large to fit in the address space of the 16 bit parts.

```
FATAL: Bad record format on input file: filename
```

This error message is generated whenever a record in a input file doesn't conform to a legal record type. This indicates that the named file has been corrupted, or that a change made to the record format generated by TASM or TLIB has not been made to TLNK (for those who can't help doing a little "improving" to programs now and then).

```
FATAL: Bad record format on library file: filename
```

This error message is generated whenever a record in a library file doesn't conform to a legal record type. This indicates that the indicated file has been corrupted, or that a change made to the record format generated by TASM or TLIB has not been made to TLNK (for those who can't help doing a little "improving" to programs now and then).

```
FATAL: Command filename too long
```

The file pathname you provided on the TLNK command line is too long. As released, TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h").

`FATAL: Corrupted temporary file`

This error usually occurs when the contents of a temporary file get corrupted by the file system somehow. If you have been changing TLNK or recompiling it for another system, this error message indicates that the "type" field in one of the internal temporary file records was not one of the allowed types. This generally happens when you make a change to one of the places which adds or removes temporary file records without changing all the other occurrences.

`FATAL: Error reading command file: filename`

TLNK got an error return while reading the specified command file. This usually indicates trouble with whatever mass storage device is being used, and/or a corrupted input file.

`FATAL: Error reading input file: filename`

TLNK got an error return during one of its read operations on input file "filename". This usually indicates trouble with whatever mass storage device is being used, and/or a corrupted input file.

`FATAL: Error reading temp file on current directory`

TLNK got an error return during one of its read operations on one of the temporary files being used (on the current directory). This error usually indicates trouble with whatever mass storage device is being used for the temporary files.

`FATAL: Error reading temp file on path: pathname`

TLNK got an error return during one of its read operations on one of the temporary files being used. The "pathname" shows which directory the temporary file is on. This error usually indicates trouble with whatever mass storage device is being used for the temporary files.

`FATAL: Error seeking file: filename`

This happens when TLNK gets a error return when attempting to do a "fseek" on the specified file. This generally indicates file system trouble.

`FATAL: Error setting stream buffer for file: filename`

This error results when TLNK is compiled with a non-zero IOBUFSIZE in file "taldef.h" but is unable to explicitly set the temporary file I/O buffer using "setvbuf" during execution. The return code from the "setvbuf" call is what actually triggers this error. As a workaround you can set IOBUFSIZE to 0 and recompile TLNK, or you can figure out what is wrong with your "C" library. The file listed is the temporary file to which TLNK was attempting to attach the buffer.

`FATAL: Error writing output file: filename`

TLNK detected an error while it was writing the linked output file. This error generally occurs when insufficient disk space is available for the output file, as well as the temporary files which also exist during this period.

`FATAL: Error writing temp file on current directory`

At some point TLNK was unable to write to a temporary file on the current directory. This generally occurs because of insufficient space on whatever device the temporary files are being written on.

`FATAL: Error writing temp file on path: pathname`

At some point TLNK was unable to write to a temporary file on the named directory. This generally occurs because of insufficient space on whatever device the temporary files are being written on.

`FATAL: Global symbol table full`

The global (also called "external"), symbol table can hold a fixed number of external symbols. As released, TLNK allows 2048 entries (controlled by MAX_GLOBALS in "tlnkdef.h").

`FATAL: Input file has already been linked: filename`

TLNK doesn't allow files which have already been linked to be re-linked. Linked files have a default filename extension of ".tld". This error is detected by examining tag data in the input file to determine file type (thus, this error could also indicate a corrupted input file).

`FATAL: Insufficient global symbol table string memory`

TLNK was unable to obtain (via "malloc" calls), enough memory to hold the external symbol names used in the various input and library files. The obvious solution is to reduce the number and length of the symbols in the input files. Another solution is to reduce the size of the "module" temp file buffer by decreasing the size of IOBUFSIZE in "taldef.h" and recompiling TLNK (this WILL slow TLNK execution speed, however). If you are using TLNK on a PC you should eliminate unnecessary memory resident programs as a first step.

`FATAL: Insufficient local buffer memory`

TLNK allocates two buffers (via "malloc"), for explicit I/O buffering. Unlike the "stream" buffering (controlled by IOBUFSIZE using "setvbuf" calls), this memory is manually managed by TLNK to optimize I/O throughput. The size of these buffers is controlled by the "LOCAL_BUF_SIZE" macro in "taldef.h". This error message indicates that there wasn't enough memory for both of these buffers in addition to whatever system buffering has been added (via IOBUFSIZE). In general, since this buffering is more effective than the vanilla system buffering, you can try reducing (perhaps to zero), the value of IOBUFSIZE to free up some memory in the system heap. If you are using TLNK on a PC, you can also try getting rid of any unnecessary memory resident programs.

`FATAL: Insufficient memory for global symbol table`

When TLNK has been compiled under MPW on the Apple Macintosh it must dynamically allocate the local and global symbol tables (via "calloc" calls), since MPW doesn't permit more than 32K of statically allocated memory per program unit. This error message results when the "calloc" call fails due to insufficient memory for the "global" symbol table (which is allocated BEFORE the request for the local symbol table). The size of the symbol table is controlled by the "MAX_GLOBALS" macro in "tlnkdef.h".

`FATAL: Insufficient memory for local symbol table`

When TLNK has been compiled under MPW on the Apple Macintosh it must dynamically allocate the local and global symbol tables (via "calloc" calls), since MPW doesn't permit more than 32K of statically allocated memory per program unit. This error message results when the "calloc" call fails due to insufficient memory for the "local" symbol table (which is allocated AFTER the request for the global symbol table). The size of the symbol table is controlled by the "MAX_SYMBOLS" macro in "tlnkdef.h".

`FATAL: Insufficient stream buffer memory for file: filename`

If the value of IOBUFSIZE in "taldef.h" is non-zero, TLNK will explicitly allocate temporary file I/O buffers (via "malloc" calls). If the memory can't be obtained for one of these buffers, this error message results. The filename listed is the one for which the buffer was intended. To get around this problem you should try to increase the amount of available "C" heap memory. If you are using TLNK on a PC, get rid of any unnecessary memory resident programs. As a last ditch effort you can reduce the value of IOBUFSIZE and recompile TLNK, but TLNK execution speed will suffer noticeably.

`FATAL: Invalid flag character: character`

The specified character wasn't recognized by TLNK as one of the option flags it supports. This error message happens when you have an error in a linker command file associated with the "FLAG" option.

`FATAL: Libraries aren't legal input files: filename`

The specified library file was named in a input file list. TLNK only allows normal relocatable files to be listed there (files with default extensions of ".trl"). This error is detected by examining tag data in the input file to determine file type (thus, this error could also indicate a corrupted input file).

`FATAL: Library path filename too long: filename`

The combination of the listed library filename and the library search path specified in the "LIB" environment variable was too long. As released TLNK allows pathnames to be 80 characters long (controlled by the FNSIZE definition in "taldef.h").

`FATAL: Local symbol table configuration error`

This is generated by an "impossible" source configuration error caused by recompiling TLNK to run under MS-DOS with an improperly modified local symbol table size. For more information, see the comments in the "tlnk1.c" source file where this error is detected.

`FATAL: Not a library file: filename`

The specified "library" file does not contain the correct tag information which indicates it is really a library. This could indicate a corrupted file, but usually reflects typing the wrong filename on a library file input specification list.

`FATAL: Out of vmfile memory`

This error can only happen if you are running TLNK configured for virtual memory temporary file buffers instead of external file system temporary files. This is the standard "too little memory available" message. If your system supports virtual memory you probably won't see this, if you are running under MS-DOS you may see it more than you might desire (thus the disk file based version of TLNK).

`FATAL: The size of SLONG is not correctly configured`

This error message can only appear when you are recompiling TLNK. It indicates that the "typedef" for SLONG which appears in "taldef.h" is set for a storage class which is less than 4 bytes long. The SLONG storage class MUST be signed for TLNK to operate correctly.

`FATAL: TLNK internal error #XXX`

These errors should never occur! If one does it generally indicates a violation of one or more prefix optimization "constraints". If this error message does occur, please send a machine-readable copy of the offending TLNK input files (and libraries), together with a description of what command line switches were used to either Logical Systems or your dealer. Be sure to indicate what operating system TLNK was running under and the complete text of the resulting error message (plus any other information you feel is pertinent). As a workaround, you can try adding, deleting or moving around bits of code in your program to see if you can avoid the exact sequence of optimization steps which provoked the problem.

`FATAL: Too many total symbols`

The total (also called "local"), symbol table can hold a fixed number of local and external symbols. As released, TLNK allows 5450 entries (controlled by MAX_SYMBOLS in "tlnkdef.h"). If TLNK is not running on a PC then MAX_SYMBOLS can probably be greatly increased (assuming a reasonable DIRECT address range). On a PC MAX_SYMBOLS can also be increased but it will force use of a mixed memory programming model and will affect the execution speed of TLNK.

`FATAL: Unable to close input file: filename`

For some reason the specified input file couldn't be closed. This generally indicates a problem with the file system.

`FATAL: Unable to close library file: filename`

For some reason the specified library file couldn't be closed. This generally indicates a problem with the file system.

`FATAL: Unable to close listing file: filename`

TLIB was unable to close the specified listing file. This is detected when TLNK is recovering from an input file specification error and is attempting to "reset" its state so that a corrected input file specification may be accepted.

`FATAL: Unable to close temp file: filename`

TLIB was unable to close the specified temporary file. This is detected when TLNK is recovering from an input file specification error and is attempting to "reset" its state so that a corrected input file specification may be accepted.

`FATAL: Unable to mix code for different CPU types`

TLNK is unable to mix code assembled for different Transputer types (T414B, T800, etc). TLIB uses information provided in the various input and library files to ensure that all the code involved is for the same type of processor.

`FATAL: Unable to open command file: filename`

TLNK was unable to open the command file you specified on the command line. You should ensure that the filename (and associated optional directory path), were correctly typed. Also note that the default filename extension is ".lnk" unless you explicitly specify one.

`FATAL: Unable to open input file: filename`

TLNK was unable to open the specified input filename. You should ensure that the filename (and associated optional directory path), were correctly typed in the command file. Also note that the default filename extension is ".trl" unless you explicitly specify one.

`FATAL: Unable to open output file: filename`

This error happens when TLNK was unable to open/create a output file for the linked program. This usually means the desired directory is write-protected, the file system is full, etc.

`FATAL: Unable to open library file: filename`

TLNK was unable to open the specified library file. This usually means that you made a mistake in the library filename or the directory path for the library. Another possible cause is a library which doesn't use the default filename extension of ".tll", and for which you failed to explicitly specify an extension.

`FATAL: Unable to open listing file: filename`

This error happens when TLNK was unable to open/create a listing file. This usually means the desired directory is write-protected, the file system is full, etc.

`FATAL: Unable to open temp file: filename`

The open attempt for the temporary "filename" failed. This filename includes whatever directory pathname was specified for temporary files.

`FATAL: Unable to remove listing file: filename`

TLIB was unable to remove the specified listing file. This is detected when TLNK is recovering from an input file specification error and is attempting to "reset" its state so that a corrected input file specification may be accepted.

```
FATAL: Unable to remove temp file: filename
```

TLIB was unable to remove the specified temporary file. This is detected when TLNK is recovering from an input file specification error and is attempting to "reset" its state so that a corrected input file specification may be accepted.

```
FATAL: Unexpected end of vmfile
```

This error can only happen if you are running TLNK configured for virtual memory temporary file buffers instead of external file system temporary files. Basically it means that TLNK encountered a premature end of file on an internal temporary file. This would indicate either a hardware or software memory allocation problem or a software failure in TLNK.

```
FATAL: Usage is 'tlnk [command_file]'
```

This error message is generated if the command line is incorrectly formatted (more than the one allowed command file as a parameter).

```
WARNING: Unable to close command file: filename
```

TLNK was unable to close the specified command file. This is done prior to the opening of the TLNK output file in an effort to minimize the number of open files.

```
WARNING: Unable to close temp file on current directory
```

During the cleanup process, prior to TLNK terminating, the temporary files are closed and deleted. This message indicates that TLNK got an error return when it attempted to close one of the temporary files on the current directory.

```
WARNING: Unable to close temp file on path: pathname
```

During the cleanup process, prior to TLNK terminating, the temporary files are closed and deleted. This message indicates that TLNK got an error return when it attempted to close one of the temporary files on the directory specified by "pathname".

```
WARNING: Unable to remove temp file: filename
```

During the cleanup process, prior to TLNK terminating, the temporary files are closed and deleted. This message indicates that TLNK was unable to remove the named temporary file (something is probably happening to the file system).

Appendix B: TLNK Internals

Source Code Organization and Compiling

The TLNK system consists of four "C" source files and four include files:

1. "tlnk1.c". Top level and I/O primitives.

2. "tlnk2.c". Operator and file input parsing.

3. "tlnk3.c". Prefix "binding" and optimization.

4. "tlnk4.c". Map file generation.

5. "taldef.h". Configuration include file for TLNK and the other assembly language tools in the Transputer Toolset. Contains the host operating system configuration selection logic, symbolic opcode names, etc.

6. "tlnkdef.h". Include file, for TLNK only, which defines configuration settings.

7. "tlnktyp.h". Include file for, TLNK only, which defines structure types, etc.

8. "tlnkext.h". Include file for, TLNK only, which defines external function and data types.

For MS-DOS source distributions the "makefile" file may be used with the supplied MAKE utility to build "tlnk.exe" using Microsoft "C" V6.00a or Borland C++ V2.0 (the Microsoft/Borland "C" compilers are not supplied and must be purchased separately). For Macintosh source file distributions consult the supplemental information your vendor has included with the Transputer Toolset.

The virtual memory version ("vtlnk.exe" for MS-DOS), may also be built using the "makefile" file and MAKE. The only difference when building this version over "tlnk.exe" is that a preprocessor symbol called "VMFILE" is defined on the command line when compiling the TLNK "C" source files. The end result is the "vtlnk.exe" version which uses internal memory instead of temporary files.